**Relatório Final: Análise de Algoritmos de Ordenação**

**1. Introdução**

O objetivo deste projeto foi implementar e analisar diferentes algoritmos de ordenação para entender suas características, desempenho prático e complexidade teórica. Os algoritmos estudados foram: **Bubble Sort**, **Selection Sort**, **Insertion Sort**, **Merge Sort**, **Quick Sort** e **Heap Sort**. Essa análise buscou fornecer insights sobre como cada algoritmo se comporta em diferentes cenários e tamanhos de entrada.

**2. Descrição Teórica dos Algoritmos**

1. **Bubble Sort**: Repetidamente compara e troca elementos adjacentes fora de ordem.
   * Complexidade: O(n^2)
2. **Selection Sort**: Seleciona o menor elemento e o coloca na posição correta a cada iteração.
   * Complexidade: O(n^2) em todos os casos.
3. **Insertion Sort**: Constrói a lista final ordenada inserindo elementos em sua posição correta.
   * Complexidade: O(n^2) no pior caso e O(n) no melhor caso.
4. **Merge Sort**: Divide a lista em partes menores, ordena cada parte e as mescla.
   * Complexidade: O (n log n) em todos os casos.
5. **Quick Sort**: Usa a estratégia de dividir e conquistar com base na escolha de um pivô.
   * Complexidade: O (n log n) no melhor e médio caso, O(n^2) no pior caso.
6. **Heap Sort**: Utiliza uma estrutura de heap para ordenar elementos.
   * Complexidade: O (n log n) em todos os casos.

**3. Metodologia**

Para a análise prática, foram seguidos os passos abaixo:

1. **Implementação**: Os algoritmos foram implementados em Java, utilizando arrays de inteiros.
2. **Testes de Performance**:
   * Gerados arrays aleatórios de tamanhos 100, 1.000, 10.000, 50.000 e 100.000.
   * Utilizado System.nanoTime() para medir os tempos de execução.
   * Cada teste foi executado 5 vezes, calculando-se a média dos tempos.
3. **Visualização de Dados**: Criado um gráfico comparando os tempos médios de execução dos algoritmos em função do tamanho do array.

**4. Resultados**

O gráfico abaixo mostra o desempenho prático dos algoritmos:

*(Inclua o gráfico gerado aqui.)*

**Observações:**

* Algoritmos O(n^2) (Bubble, Selection e Insertion Sort) têm tempos de execução que crescem rapidamente com o aumento do tamanho do array.
* Algoritmos O(n log n) (Merge, Quick e Heap Sort) são significativamente mais rápidos para entradas grandes.
* Quick Sort foi o mais eficiente, seguido por Merge Sort e Heap Sort.

**5. Análise Comparativa**

| **Algoritmo** | **Complexidade Teórica** | **Desempenho Prático** | **Situações Ideais** |
| --- | --- | --- | --- |
| Bubble Sort | O(n^2) | Lento para entradas grandes | Pequenas entradas ordenadas ou aprendizado. |
| Selection Sort | O(n^2) | Mais rápido que Bubble | Entrada pequena sem necessidade de estabilidade. |
| Insertion Sort | O(n) (melhor caso) | Eficiente para entradas pequenas ordenadas. | Quase ordenado ou pequenas listas. |
| Merge Sort | O(n log n) | Consistente e estável | Entradas grandes que necessitam de estabilidade. |
| Quick Sort | O(n log n) (geral) | Mais rápido em média | Entradas grandes e desordenadas. |
| Heap Sort | O(n log n) | Consistente, mas menos eficiente que Quick. | Quando a memória é uma restrição. |

**6. Conclusão**

Este projeto demonstrou a importância de escolher algoritmos de ordenação de acordo com o contexto e o tamanho da entrada. Enquanto algoritmos simples podem ser viáveis em situações específicas, algoritmos mais avançados como Quick Sort e Merge Sort são essenciais para lidar com grandes volumes de dados. A análise prática confirmou a relevância das previsões teóricas de complexidade.

**7. Referências**

1. **Cormen, T. H. et al.** Introduction to Algorithms. MIT Press, 2009.
2. Documentação oficial Java: <https://docs.oracle.com/javase/>

**8. Link do Repositório**

O código-fonte completo está disponível no GitHub: [Link para o repositório](https://github.com/seu_usuario/algoritmos-ordenacao).